Monad Macros in Common Lisp

David Sorokin david.sorokin@gmail.com, Jan 2010

Contents

Introduction

A monad can be defined with help of two functions, one of which is higher-order. Direct working with them is tedious and error-prone. In this article I'll describe an approach that greatly simplifies the use of monads in Common Lisp. It is possible due to macros.

I suppose that the reader is familiar with Haskell's definition of the Monad type class. To create a monad instance, we have to define the mentioned two functions. The first of them is called *return*. The second one is known as the *bind* function and it is denoted in Haskell as operator (>>=):

```
class Monad m where
      return :: a -> m a
      (>>=)  :: m a -> (a -> m b) -> m b
```

This definition actually allows the programmer to use common names return and (>>=) for very different functions. I'll try to create similar Lisp macros that will be common for all monads. Also Haskell provides a useful *do*-notation which is a syntactic sugar for monads. The macros I will create will provide similar facilities as well.

Also I created a new project with name **cl-monad-macros**. It is available by the following link: http://common-lisp.net/project/cl-monad-macros. The corresponded package contains definitions of all monad macros described in this article.

The package and all examples were successfully tested on the following Lisp systems:

- Steel Bank Common Lisp (SBCL);
- Clozure CL (CCL);
- CLISP;
- LispWorks;
- Allegro CL.

General Case

Let's suppose that some monad is defined with help of two hypothetical functions UNITF and FUNCALLF:

```
(defun unitf (a)
      ;; evaluate as in Haskell: return a
      …)

(defun funcallf (k m)
      ;; evaluate as in Haskell: m >>= k
      …)
```

The UNITF function is the return function. Function FUNCALLF is an analog of the idiomatic bind function but only the order of arguments is opposite. Further I call namely this new function a *bind* function. Please take care.

We could limit ourselves to using only these functions, but it would be tedious. Please take into account that the first argument of the bind function must be a function, most probably an anonymous function. Moreover, we can use a sequence of monad values in one computation, which complicates the matter significantly.

Therefore I offer to use the following macros:

Common Lisp	Haskell
`(unit a)`	`return a`
`(funcall! k m)`	`m >>= k`
`(progn! m1 m2 … mn)`	`m1 >> m2 >> … >> mn`

`(let! ((x1 e1)`
`(x2 e2)`
`…`
`(xn en))`
`m)`
	`do x1 <- e1`
	`x2 <- e2`
	`…`
	`xn <- en`
	`m`

The UNIT macro is equivalent to a call of the *return* function. The FUNCALL! macro is expanded to a call of the bind function. Macro PROGN! is equivalent to the monadic *then* function, which is denoted in Haskell as (>>). It allows the programmer to create a sequence of computations. Internally, it is based on more primitive FUNCALL! macro.

Source form	Reduction form
`(progn! m)`	`m`

`(progn! m1 m2)`	`(funcall!`
	`#'(lambda (#:gen-var)`
	`(declare (ignore #:gen_var))`
	`m2)`
	`m1)`

`(progn! m1 m2 … mn)`	`(progn! m1 (progn! m2 … mn))`

Here *#:gen-var* means an automatically generated unique variable name with help of GENSYM.

Macro LET! is somewhere an alternative to the arrow symbol from the *do*-notation of Haskell. It is also based on the FUNCALL! macro. It binds computations *e1*, *e2*, …, *en* with values *x1*, *x2*, …, *xn,* which can be then used in computation *m*.

Source form	Reduction form

`(let! ((x e)) m)`	`(funcall!`
	`#'(lambda (x) m)`
	`e)`

`(let! ((x1 e1)`
`(x2 e2)`
`…`
`(xn en))`
	`(let! ((x1 e1))`
	`(let! ((x2 e2)`
	`…`
	`(xn en))`

m)	*m*))

Please note that the LET! macro accepts only two arguments, the last of which is the monad value. It was made intentionally for similarity with the LET and LET* operators in the following sense. If we want to propagate a sequence of computations then we have to apply the PROGN! macro in all cases:

Common Lisp	Haskell
(let! ((*x e*)) (progn! *m1 m2 … mn*))	do *x* <- *e* *m1* *m2* … *mn*
(let ((*x a*)) (progn! *m1 m2 … mn*))	do let *x* = *a* *m1* *m2* … *mn*

Thus, macros UNIT, FUNCALL!, PROGN! and LET! provide an unified common way of working with the monads. To distinguish different monads from each other, we can implement these macros as a MACROLET defined by global macro WITH-MONAD that has the following application form:

```
(with-monad (return-func funcall-func)
      ;; Here we can use UNIT, FUNCALL!, PROGN! and LET!
      body1 … bodyN)
```

The first sub-parameter *return-func* defines a name of the return function. The second sub-parameter *funcall-func* defines a name of the bind function. This macro is expanded to a MACROLET saving the same body.

```
(defmacro with-monad ((unit-func funcall-func) &body body)
  `(macrolet
       ((unit (a) (list ',unit-func a))
        (funcall! (k m) (list ',funcall-func k m))
        (progn! (&body ms) (append '(generic-progn!) '(,funcall-func) ms))
        (let! (decls m) (list 'generic-let! ',funcall-func decls m)))
     ,@body))
```

Here the GENERIC-LET! macro is used to process the LET! expression in accordance with the stated above definition.

```
(defmacro generic-let! (funcall-func decls m)
  (reduce #'(lambda (decl m)
              (destructuring-bind (x e) decl
                `(,funcall-func #'(lambda (,x) ,m) ,e)))
          decls
          :from-end t
          :initial-value m))
```

The PROGN! expression is processed already by the GENERIC-PROGN! helper macro.

```
(defmacro generic-progn! (funcall-func &body ms)
  (reduce #'(lambda (m1 m2)
              (let ((x (gensym)))
                `(,funcall-func
                  #'(lambda (, x)
                      (declare (ignore ,x))
                      ,m2)
                  ,m1)))
          ms
          :from-end t))
```

Then the following test expression

```
  (with-monad (unitf funcallf)
    (let! ((x1 e1)
           (x2 e2))
          (progn! m1 m2
                  (unit (list x1 x2)))))
```

is expanded ultimately to

```
  (FUNCALLF
   #'(LAMBDA (X1)
       (FUNCALLF
        #'(LAMBDA (X2)
            (FUNCALLF
             #'(LAMBDA (#:G983)
                 (DECLARE (IGNORE #:G983))
                 (FUNCALLF
                  #'(LAMBDA (#:G982)
                      (DECLARE (IGNORE #:G982))
                      (UNITF (LIST X1 X2)))
                  M2))
             M1))
        E2))
   E1)
```

The expanded code is generic enough. Actually, macro WITH-MONAD satisfies some abstract contract providing definitions for macros UNIT, FUNCALL!, PROGN! and LET!. As we'll see later, there are other specialized macros that are like WITH-MONAD and that satisfy the same contract but generate a more efficient code for their monads. Moreover, in case of the monad transformers new macros are necessary.

Monad	Monad Macro
General Case	WITH-MONAD
The Identity Monad	WITH-IDENTITY-MONAD
The List Monad	WITH-LIST-MONAD
The Maybe Monad	WITH-MAYBE-MONAD
The Reader Monad	WITH-READER-MONAD
The State Monad	WITH-STATE-MONAD

The Writer Monad	WITH-WRITER-MONAD
The Reader Monad Transformer	WITH-READER-MONAD-TRANS
The State Monad Transformer	WITH-STATE-MONAD-TRANS
The Writer Monad Transformer	WITH-WRITER-MONAD-TRANS

It's important that macros like WITH-MONAD can be nested, which allows the programmer to work with different monads in the same s-expression. Each new application of the WITH-MONAD macro shadows the previous definition of macros UNIT, FUNCALL!, PROGN! and LET!. It means that at any moment only one monad can be active.

Although we can always use directly the WITH-MONAD macro, it is more convenient to create a short name for each monad in accordance with the following pattern:

```
(defmacro with-my-monad (&body body)
   `(with-monad (unitf funcallf)
      ,@body))
```

where UNITF and FUNCALLF were used as an example.

Bind Macros

In the rest of the article you'll see a lot of definitions of the LET! and PROGN! macros. Actually, all them can be reduced to the following two macros that will work with any monad.

```
(defmacro universal-progn! (&body ms)
  (reduce #'(lambda (m1 m2)
              (let ((x (gensym)))
                `(funcall!
                  #'(lambda (,x)
                      (declare (ignore ,x))
                      ,m2)
                  ,m1)))
          ms
          :from-end t))

(defmacro universal-let! (decls m)
  (reduce #'(lambda (decl m)
              (destructuring-bind (x e) decl
                `(funcall! #'(lambda (,x) ,m) ,e)))
          decls
          :from-end t
          :initial-value m))
```

Nevertheless, there is one subtle optimization issue related to the order of arguments of the FUNCALL! macro. During the macro expansion of expression

```
  (let! ((x e)) m)
```

macro UNIVERSAL-LET! will generate ultimately for the most of monads described in this article something like

```
(let ((k #'(lambda (x) m)))     ; save the first argument of FUNCALL!
  …
  (let ((a (f e)))              ; use the second argument of FUNCALL!
    (funcall k a))
  …)
```

But I'm not sure that any Lisp compiler is able to optimize it to the following equivalent code that would be more efficient

```
…
(let ((x (f e)))
  m)
…
```

Please note that there would be no such problem if the FUNCALL! macro had another order of parameters, i.e. an idiomatic order as in Haskell. Then FUNCALL and LAMBDA would alternate with each other directly in the code and the compiler most probably could reduce them.

```
…
(let ((a (f e)))
  (funcall
    #'(lambda (x) m)
    a))
…
```

But I think that a similarity with the standard FUNCALL function is more important and I'm ready to provide optimized versions of the LET! and PROGN! macros whenever it makes sense.

The Identity Monad

The Identity monad is the simplest case. The return function is IDENTITY. The bind function is FUNCALL. Then UNIT macro becomes an acronym of the IDENTITY function, FUNCALL! becomes the ordinary FUNCALL, PROGRN! is equivalent to PROGN, but LET! is transformed to LET*. This coincidence in names can be considered as a rule of thumb. Only the LET! macro is a small exception.

```
(defmacro with-identity-monad (&body body)
  `(with-monad (identity funcall)
     ,@body)
```

But there is a much more efficient implementation:

```
(defmacro with-identity-monad (&body body)
  `(macrolet
       ((unit (a) a)
        (funcall! (k m) (list 'funcall k m))
        (progn! (&body ms) (append '(progn) ms))
        (let! (decls m) (list 'let* decls m)))
     ,@body))
```

Remembering about this monad, it is easy to memorize names FUNCALL!, PROGN! and LET!.

Our test expression

```
  (with-identity-monad
    (let! ((x1 e1)
           (x2 e2))
          (progn! m1 m2
                  (unit (list x1 x2)))))
```

is expanded to

```
  (LET* ((X1 E1) (X2 E2))
    (PROGN M1 M2 (LIST X1 X2)))
```

The List Monad

This section is devoted to the List monad. I'll introduce macro WITH-LIST-MONAD that will implement a contract of the WITH-MONAD macro but that will do it in its own optimized way.

A monad value is just a list. Following the idiomatic definition, we can write the UNIT and FUNCALL! macro prototypes:

```
(defmacro list-unit (a)
   `(list ,a))

(defmacro list-funcall! (k m)
   `(reduce #'append (mapcar ,k ,m)))
```

Please note that NIL is also a value of the list monad. We'll use this fact further.

Here is a definition of the PROGN! macro prototype.

```
(defmacro list-progn! (&body ms)
  (reduce
     #'(lambda (m1 m2)
          (let ((x (gensym)))
             `(loop for ,x in ,m1 append ,m2)))
     ms
     :from-end t))
```

At each reduction step we introduce a loop that appends the second argument as many times as the length of the first list. If the first list is NIL then the result of the loop is NIL as well.

The LET! macro prototype can be implemented similarly and also without use of the lambda.

```
(defmacro list-let! (decls m)
  (reduce
     #'(lambda (decl m)
          (destructuring-bind (x e) decl
             `(loop for ,x in ,e append ,m)))
     decls
     :from-end t
     :initial-value m))
```

Here we replace each variable binding with the corresponded loop. It should generate an efficient enough code.

Macros UNIT, FUNCALL!, PROGN! and LET! actually are defined in a MACROLET implemented by the WITH-LIST-MONAD macro.

```
(defmacro with-list-monad (&body body)
  `(macrolet
       ((unit (a) `(list ,a))
        (funcall! (k m) `(reduce #'append (mapcar ,k ,m)))
        (progn! (&body ms) (append '(list-progn!) ms))
        (let! (decls m) (list 'list-let! decls m)))
     ,@body))
```

The same test example

```
  (with-list-monad
    (let! ((x1 e1)
           (x2 e2))
          (progn! m1 m2
                  (unit (list x1 x2)))))
```

is now expanded to

```
(LOOP FOR X1 IN E1
   APPEND (LOOP FOR X2 IN E2
     APPEND (LOOP FOR #:G1030 IN M1
       APPEND (LOOP FOR #:G1029 IN M2
         APPEND (LIST (LIST X1 X2))))))
```

We can ask for something more practical:

```
CL-USER> (with-list-monad
            (let ((numbers '(1 2 3 4 5 6 7 8 9 10)))
                 (let! ((x numbers)
                        (y numbers)
                        (z numbers))
                       (if (= (+ (* x x) (* y y)) (* z z))
                           (unit (list x y z))))))

((3 4 5) (4 3 5) (6 8 10) (8 6 10))
```

Please note that here we use the fact that NIL is a legal value of the List monad. Therefore we can omit the else-part of the IF operator. Moreover, if *numbers* were an empty list then the topmost loop would immediately return NIL.

Also we can define the following function *perms* that produces a list of permutations of a given list.

```
(defun perms (xs)
  (with-list-monad
      (if (null xs)
          (unit nil)
          (let! ((y xs)
```

```
                  (ys (perms (remove y xs :count 1))))
                 (unit (cons y ys))))))
```

Now we can test it.

```
CL-USER> (perms '(1 2 3))
((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))
```

The Maybe Monad

The next monad is the Maybe monad. It allows efficiently stopping a complex sequence of computations right after discovering a failure. If there is no failure then a full chain of computations is performed.

The constructor, getters and predicates for this data type are defined below.

```
(defmacro make-maybe (&key (just nil just-supplied-p))
  (if just-supplied-p `(cons ,just nil)))

(defmacro maybe-just (a)
  `(car ,a))

(defmacro maybe-nil ()
  nil)

(defmacro maybe-just-p (m)
  `(consp ,m))

(defmacro maybe-nil-p (m)
  `(null ,m))
```

The prototypes of the basic return and bind macros can be defined in the following way.

```
(defmacro maybe-unit (a)
  `(make-maybe :just ,a))

(defmacro maybe-funcall! (k m)
  (let ((xk (gensym))
        (xm (gensym)))
    `(let ((,xk ,k)
           (,xm ,m))
       (if (maybe-nil-p ,xm)
           (make-maybe)
           (funcall ,xk (maybe-just ,xm))))))
```

The key point is the IF expression that cuts the further computation if the result of the former one is NIL.

Based on these macros we can build their counterpart PROGN!.

```
(defmacro maybe-progn! (&body ms)
  (reduce
     #'(lambda (m1 m2)
         `(if (maybe-nil-p ,m1)
              (make-maybe)
```

```
                ,m2))
     ms
     :from-end t))
```

The LET! macro is similar but it allows the programmer to bind variables within one computation.

```
(defmacro maybe-let! (decls m)
  (reduce
    #'(lambda (decl m)
        (destructuring-bind (x e) decl
          (let ((xe (gensym)))
            `(let ((,xe ,e))
               (if (maybe-nil-p ,xe)
                   (make-maybe)
                   (let ((,x (maybe-just ,xe)))
                        ,m))))))
    decls
    :from-end t
    :initial-value m))
```

In the three cases we see the cutting IF expressions. They stop immediately the computation right after discovering a failure.

Actually, these last four macros are implemented as a MACROLET defined by macro WITH-MAYBE-MONAD. As always, we could implement the latter with help of generic macro WITH-MONAD providing the necessary return and bind functions which are trivial for this monad. But macro WITH-MAYBE-MONAD is much more efficient.

```
(defmacro with-maybe-monad (&body body)
  `(macrolet
       ((unit (a) (list 'maybe-unit a))
        (funcall! (k m) (list 'maybe-funcall! k m))
        (progn! (&body ms) (append '(maybe-progn!) ms))
        (let! (decls m) (list 'maybe-let! decls m)))
     ,@body))
```

Our old example

```
  (with-maybe-monad
    (let! ((x1 e1)
           (x2 e2))
          (progn! m1 m2
                  (unit (list x1 x2)))))
```

is expanded to

```
  (LET ((#:G1051 E1))
    (IF (NULL #:G1051) NIL
        (LET ((X1 (CAR #:G1051)))
          (LET ((#:G1050 E2))
            (IF (NULL #:G1050) NIL
                (LET ((X2 (CAR #:G1050)))
```

```
                    (IF (NULL M1) NIL
                        (IF (NULL M2) NIL (CONS (LIST X1 X2) NIL)))))))))
```

Now we can consider something more illustrative

```
CL-USER> (with-maybe-monad
           (progn! (progn
                     (format t "Step 1.")
                     (make-maybe :just 'OK))
                   (make-maybe)  ; NIL – failure
                   (progn
                     (format t "Step 2.")
                     (make-maybe :just 'OK))))

Step 1.
NIL
```

Moreover, SBCL will warn about an unreachable code during compilation if we'll try to define such a function!

The Reader Monad

The Reader monad is a rather complicated thing. The monad value is a function that returns a result of the computation by the given environment value. In Haskell it can be defined like this

```
import Control.Monad

newtype Reader r a = Reader {runReader :: r -> a}

instance Monad (Reader r) where

    return a = Reader (\r -> a)

    m >>= k = Reader (\r ->
                          let a  = runReader m r
                              m' = k a
                          in runReader m' r)

read :: Reader r r
read = Reader (\r -> r)
```

In accordance with this definition I'll create a monad macro WITH-READER-MONAD.

The UNIT macro prototype is simple enough.

```
(defmacro reader-unit (a)
  (let ((r (gensym)))
    `#'(lambda (,r)
         (declare (ignore ,r))
         ,a)))
```

The FUNCALL! macro prototype is crucial for understanding the monad macro.

```
(defmacro reader-funcall! (k m)
  (let ((r (gensym))
        (a (gensym))
        (kg (gensym)))
    `#'(lambda (,r)
         (let ((,kg ,k)
               (,a (funcall ,m ,r)))
           (funcall (funcall ,kg ,a) ,r)))))
```

There is a subtle thing. Parameter *k* is evaluated inside the anonymous function returned. In other words, its evaluation is delayed. I think that the user will expect namely such a behavior. Moreover, it allows the Lisp compiler to optimize the code in case of the PROGN! and LET! macros as it will be shown.

Also please note that value *m*, being a monad value, is actually an anonymous function. If its s-expression will be accessible during the macro expansion then we'll receive something similar to

```
  (funcall #'(lambda (x) f) r)
```

which can be efficiently optimized by the compiler to

```
  (let ((x r)) f)
```

The LET! macro prototype is more efficient than FUNCALL! as one of the FUNCALLs becomes unnecessary.

```
(defmacro reader-let! (decls m)
  (reduce #'(lambda (decl m)
              (destructuring-bind (x e) decl
                (let ((r (gensym)))
                  `#'(lambda (,r)
                       (let ((,x (funcall ,e ,r)))
                         (funcall ,m ,r))))))
          decls
          :from-end t
          :initial-value m))
```

Here like expression *e* expression *m* is evaluated inside FUNCALL. It's also a monad value, i.e. an anonymous function. If we'll create a LET! expression with many variable bindings then the s-expression of *m* will be accessible during the macro expansion for all bindings but probably the last. It will allow the Lisp compiler to optimize the LET! expression essentially. We'll see an example in the end of this section.

The PROGN! macro prototype is more simple as we don't bind variables.

```
(defmacro reader-progn! (&body ms)
  (reduce #'(lambda (m1 m2)
              (let ((r (gensym)))
                `#'(lambda (,r)
                         (funcall ,m1 ,r)
                         (funcall ,m2 ,r))))
          ms
          :from-end t))
```

Again, if the s-expression for *m1* and *m2* will be accessible then the Lisp compiler will have good chances to generate a more optimal code.

The Reader monad was created for one purpose – to pass some value through all the computations. Let it be macro READ! that gets this value and puts in the monad. It corresponds to the *read* value defined above in Haskell. The macro prototype is as follows.

```
(defmacro reader-read! ()
  (let ((r (gensym)))
    `#'(lambda (,r) ,r)))
```

A computation in the Reader monad must be started somewhere. We take some value and pass it to the computation. This monad computation is passed in the first parameter. The environment value is passed in the second parameter. The corresponded macro has name RUN! and its prototype is defined below.

```
(defmacro reader-run! (m r)
  `(funcall ,m ,r))
```

The value returned is a result of the monad computation.

Macros READ!, RUN!, UNIT, FUNCALL!, PROGN! and LET! are implemented as a MACROLET defined by the WITH-READER-MONAD macro.

```
(defmacro with-reader-monad (&body body)
  `(macrolet
       ((unit (a) (list 'reader-unit a))
        (funcall! (k m) (list 'reader-funcall! k m))
        (progn! (&body ms) (append '(reader-progn!) ms))
        (let! (decls m) (list 'reader-let! decls m))
        (read! () (list 'reader-read!))
        (run! (m r) (list 'reader-run! m r)))
     ,@body))
```

Now we can take our old test example

```
  (with-reader-monad
    (let! ((x1 e1)
           (x2 e2))
          (progn! m1 m2
                  (unit (list x1 x2)))))
```

and look at the result of the macro expansion.

```
  #'(LAMBDA (#:G788)
      (LET ((X1 (FUNCALL E1 #:G788)))
        (FUNCALL
         #'(LAMBDA (#:G787)
             (LET ((X2 (FUNCALL E2 #:G787)))
               (FUNCALL
                #'(LAMBDA (#:G790)
                    (FUNCALL M1 #:G790)
                    (FUNCALL
```

```
                      #'(LAMBDA (#:G789)
                          (FUNCALL M2 #:G789)
                          (FUNCALL
                           #'(LAMBDA (#:G791)
                               (DECLARE (IGNORE #:G791))
                               (LIST X1 X2))
                           #:G789))
                      #:G790))
                  #:G787)))
          #:G788)))
```

We can see that there are many LAMBDAs and FUNCALLs bound together. A good Lisp compiler must generate a rather efficient code.

Here is a small test

```
(defun reader-test ()
  (with-reader-monad
    (run!
     (let! ((x (read!)))
           (progn
             (format t "x=~a~%" x)
             (unit 'ok)))
     10)))
```

and this is its output.

```
CL-USER> (reader-test)
x=10
OK
```

The State Monad

The State monad allows us to manage some state during a computation. We can put a new value or request for the current value of the state.

I'll use the next definition written in Haskell.

```
import Control.Monad

newtype State st a = State {runState :: st -> (a, st)}

instance Monad (State st) where

    return a = State (\st -> (a, st))

    m >>= k = State (\st ->
                         let (a, st') = runState m st
                             m' = k a
                          in runState m' st')

get :: State st st
get = State (\st -> (st, st))
```

```
put :: st -> State st ()
put st' = State (\_ -> ((), st'))
```

I'll create the corresponded monad macro WITH-STATE-MONAD. It will define macros GET!, PUT! and RUN! as a part of its MACROLET definition. The GET! macro will correspond to the *get* function. The PUT! macro will be an analog of the *put* function. The RUN! macro will play a role of the *runState* function.

First of all, I define utility macros.

```
(defmacro make-state (a st)
  `(cons ,a ,st))

(defmacro state-value (m)
  `(car ,m))

(defmacro state-state (m)
  `(cdr ,m))
```

The UNIT macro prototype is simple.

```
(defmacro state-unit (a)
  (let ((st (gensym)))
    `#'(lambda (,st)
         (make-state ,a ,st))))
```

Please note that we evaluate *a* inside LAMBDA, i.e. the evaluation is delayed until the anonymous function is called. I'll apply this strategy to all macros for this monad. In other words, any computation in this monad does nothing until it is explicitly started with help of macro RUN!, which will be defined further. By the way, the same strategy was true for the Reader monad.

The FUNCALL! macro prototype follows the definition of the bind function.

```
(defmacro state-funcall! (k m)
  (let ((st (gensym))
        (p (gensym))
        (a (gensym))
        (kg (gensym)))
    `#'(lambda (,st)
         (let ((,kg ,k))
           (let ((,p (funcall ,m ,st)))
             (let ((,a (state-value ,p)))
               (funcall (funcall ,kg ,a)
                        (state-state ,p))))))))
```

All notes that I did for the FUNCALL! macro of the Reader monad are applicable here. Being a monad value, expression *m* is actually an anonymous function. If its s-expression is available at the time of macro expansion then the corresponded FUNCALL and LAMBDA can be reduced by the smart compiler.

The LET! macro prototype generates a more optimal code than FUNCALL!.

```
(defmacro state-let! (decls m)
  (reduce #'(lambda (decl m)
              (destructuring-bind (x e) decl
                (let ((st (gensym))
                      (p (gensym)))
                  `#'(lambda (,st)
                       (let ((,p (funcall ,e ,st)))
                         (let ((,x (state-value ,p)))
                           (funcall ,m (state-state ,p))))))))
          decls
          :from-end t
          :initial-value m))
```

If we create a multi-level LET! expression then *m* will be expanded to the LAMBDA expression in all cases but probably the last. It will allow the Lisp compiler to optimize the expanded code as you will see later in the example.

The PROGN! macro prototype is more simple.

```
(defmacro state-progn! (&body ms)
  (reduce #'(lambda (m1 m2)
              (let ((st (gensym))
                    (p (gensym)))
                `#'(lambda (,st)
                     (let ((,p (funcall ,m1 ,st)))
                       (funcall ,m2 (state-state ,p))))))
          ms
          :from-end t))
```

To start a computation in the State monad, we can use the RUN! macro which accepts two arguments. The first argument specifies the computation. The second argument is an initial state. The RUN! macro returns a list of two values. The first value is the result of the computation itself. The second value of this list is a final state.

```
(defmacro state-run! (m init-st)
  (let ((p (gensym)))
    `(let ((,p (funcall ,m ,init-st)))
       (list (state-value ,p)
             (state-state ,p)))))
```

To manage the state during the computation, we can use macros GET! and PUT!. The GET! macro returns the current state wrapped in the monad.

```
(defmacro state-get! ()
  (let ((st (gensym)))
    `#'(lambda (,st)
         (make-state ,st ,st))))
```

The PUT! macro allows setting a new value for the state. This value is passed as a parameter. The macro returns NIL wrapped in the monad.

```
(defmacro state-put! (new-st)
  (let ((st (gensym)))
    `#'(lambda (,st)
         (declare (ignore ,st))
         (make-state nil ,new-st))))
```

Macros RUN!, GET!, PUT!, UNIT, FUNCALL!, LET! and PROGN! are implemented as a MACROLET defined by the WITH-STATE-MONAD macro.

```
(defmacro with-state-monad (&body body)
  `(macrolet
       ((unit (a) (list 'state-unit a))
        (funcall! (k m) (list 'state-funcall! k m))
        (progn! (&body ms) (append '(state-progn!) ms))
        (let! (decls m) (list 'state-let! decls m))
        (get! () (list 'state-get!))
        (put! (new-st) (list 'state-put! new-st))
        (run! (m init-st) (list 'state-run! m init-st)))
     ,@body))
```

For our old test example

```
  (with-state-monad
    (let! ((x1 e1)
           (x2 e2))
          (progn! m1 m2
                  (unit (list x1 x2)))))
```

the macro expansion looks like

```
  #'(LAMBDA (#:G1696)
      (LET ((#:G1697 (FUNCALL E1 #:G1696)))
        (LET ((X1 (CAR #:G1697)))
          (FUNCALL
           #'(LAMBDA (#:G1694)
               (LET ((#:G1695 (FUNCALL E2 #:G1694)))
                 (LET ((X2 (CAR #:G1695)))
                   (FUNCALL
                    #'(LAMBDA (#:G1700)
                        (LET ((#:G1701 (FUNCALL M1 #:G1700)))
                          (FUNCALL
                           #'(LAMBDA (#:G1698)
                               (LET ((#:G1699 (FUNCALL M2 #:G1698)))
                                 (FUNCALL
                                  #'(LAMBDA (#:G1702)
                                      (CONS (LIST X1 X2) #:G1702))
                                  (CDR #:G1699))))
                           (CDR #:G1701))))
                    (CDR #:G1695)))))
           (CDR #:G1697)))))
```

We can note that many LAMBDAs and FUNCALLs can be reduced. The bigger is our source expression, the more such constructs can the compiler reduce. The code should be rather cheap.

The next test enumerates items of the tree and creates a new tree, where each item is replaced with the CONS-pair consisting of the item itself and its sequence number.

```
(defun state-test (tree)
  (labels
      ((order (tree)
         (with-state-monad
           (cond ((null tree) (unit nil))
                 ((consp tree)
                  (let! ((t1 (order (car tree)))
                         (t2 (order (cdr tree))))
                    (unit (cons t1 t2))))
                 (t
                  (let! ((n (get!)))
                    (let ((new-n (+ n 1)))
                      (progn!
                       (put! new-n)
                       (unit (cons tree new-n)))))))))))
    (destructuring-bind (new-tree new-state)
        (with-state-monad
          (run! (order tree) 0))
      (format t "Item count=~a~%" new-state)
      (format t "New tree=~a~%" new-tree))))
```

Now we can launch a test.

```
CL-USER> (state-test '(((5 2) 7 4) 5 9))
Item count=6
New tree=((((5 . 1) (2 . 2)) (7 . 3) (4 . 4)) (5 . 5) (9 . 6))
NIL
```

The Writer Monad

This section describes the Writer monad. This monad allows writing a log during the computation. Then this log can be requested along with the computed result.

I will use the following definition written in Haskell.

```
import Control.Monad

newtype Writer w a = Writer (a, [w] -> [w])

runWriter :: Writer w a -> (a, [w])
runWriter (Writer (a, f)) = (a, f [])

write :: w -> Writer w ()
write w = Writer ((), \xs -> w : xs)

writeList :: [w] -> Writer w ()
writeList ws = Writer ((), \xs -> ws ++ xs)

instance Monad (Writer w) where

    return a = Writer (a, id)
```

```
    (Writer (a, f)) >>= k =
        let Writer (a', f') = k a
        in Writer (a', f . f')
```

Actually, I will use a more efficient representation of the functions. We can note that the return function uses *id*, but the bind function always creates a composition of two functions (*f . f'*). This is unnecessary. In Common Lisp we can use NIL to denote the identity function. It will be a detail of the implementation about which the user may not know. But this approach can help the compiler to generate a more efficient code in cases if the *write* and *writeList* functions are called rarely, i.e. when *f'* is just the *id* function.

I'll begin with utilities.

```
(defmacro make-writer (a fun)
  `(cons ,a ,fun))

(defmacro writer-value (m)
  `(car ,m))

(defmacro writer-fun (m)
  `(cdr ,m))
```

The next macro creates a composition of the two specified nullable functions, where NIL means the IDENTITY function.

```
(defmacro writer-compose (f g)
  ;; There are high chances that g is NIL
  (let ((fs (gensym))
        (gs (gensym)))
    `(let ((,fs ,f)
           (,gs ,g))
       (cond ((null ,gs) ,fs)    ; check it first
             ((null ,fs) ,gs)
             (t #'(lambda (x)
                    (funcall ,fs
                             (funcall ,gs x))))))))
```

Let our monad macro will have name WITH-READER-MONAD and will define three additional macros WRITE!, WRITE-LIST! and RUN!. The first two will be analogs of the *write* and *writeList* functions respectively and they will be used for writing a log. The RUN! macro will be an analog of the *runWriter* function and will be used for running a computation. The RUN! macro will return a list of two values. The first value will be a result of the computation itself. The second value will be a log written during the computation.

The WRITE! macro saves the specified values in the log. It returns NIL wrapped in the monad like that how the *write* function returns *Writer w ()*. Its prototype is as follows.

```
(defmacro writer-write! (&body ws)
    (if (= 1 (length ws))
        ;; An optimized case
        (let ((w (nth 0 ws))
```

```
              (v (gensym)))
          `(make-writer nil
                        (let ((,v ,w))
                          #'(lambda (xs) (cons ,v xs)))))
        ;; A general case
        (let ((vs (gensym)))
          `(make-writer nil
                        (let ((,vs (list ,@ws)))
                          #'(lambda (xs)
                              (append ,vs xs)))))))
```

Please note that we don't add new records. We return a function that knows how to add them. This a very efficient technique. Please compare with the *shows* function from Haskell.

The WRITE-LIST! macro prototype takes the value lists and saves their values in the log. The macro returns NIL in the monad as well.

```
(defmacro writer-write-list! (&body wss)
    (if (= 1 (length wss))
        ;; An optimized case
        (let ((ws (nth 0 wss))
              (vs (gensym)))
          `(make-writer nil
                        (let ((,vs ,ws))
                          #'(lambda (xs) (append ,vs xs)))))
        ;; A general case
        (let ((vss (gensym)))
          `(make-writer nil
                        (let ((,vss (list ,@wss)))
                          #'(lambda (xs)
                              (reduce #'append ,vss
                                      :from-end t
                                      :initial-value xs)))))))
```

The RUN! macro accepts one argument, a monad computation. It returns a list consisting of the computed value and a log written during this computation. The prototype is defined below.

```
(defmacro writer-run! (m)
  (let ((x (gensym))
        (fun (gensym)))
    `(let ((,x ,m))
       (list (writer-value ,x)
             (let ((,fun (writer-fun ,x)))
               (if (not (null ,fun))
                   (funcall ,fun nil)))))))
```

Here we take into account that the log function can be actually represented by value NIL. In such a case we return an empty list as a result log. If the function is defined then we ask it to create a log based on the initial empty log. It works fast, although the log is constructed starting from the end.

Also we can see a weakness of the method. If macros WRITE! and WRITE-LIST! were too often called then we would have a compound function consisting of a lot of nested functions. It might lead to the stack overflow. Be careful!

We consider NIL as an optimized representation of the IDENTITY function. The UNIT macro prototype uses this fact as the log remains unmodified.

```
(defmacro writer-unit (a)
  `(make-writer ,a nil))
```

The FUNCALL! macro is more complicated.

```
(defmacro writer-funcall! (k m)
  (let ((ks (gensym))
        (ms (gensym))
        (a (gensym))
        (ka (gensym)))
    `(let* ((,ks ,k)    ; save it first
            (,ms ,m)
            (,a (writer-value ,ms))
            (,ka (funcall ,ks ,a)))
       (make-writer (writer-value ,ka)
                    (writer-compose (writer-fun ,ms)
                                    (writer-fun ,ka))))))
```

As usual, based on this macro we can write a more optimal definition of the LET! macro prototype which has no FUNCALL at all.

```
(defmacro writer-let! (decls m)
  (reduce
   #'(lambda (decl m)
       (destructuring-bind (x e) decl
         (let ((es (gensym))
               (ms (gensym)))
           `(let* ((,es ,e)
                   (,x (writer-value ,es))
                   (,ms ,m))    ; depends on x!
              (make-writer (writer-value ,ms)
                           (writer-compose (writer-fun ,es)
                                           (writer-fun ,ms)))))))
   decls
   :from-end t
   :initial-value m))
```

The PROGN! macro prototype is even more simple as there is no variable binding. But we have to compose the log functions, though.

```
(defmacro writer-progn! (&body ms)
  (reduce
   #'(lambda (m1 m2)
       (let ((m1s (gensym))
             (m2s (gensym)))
         `(let ((,m1s ,m1)
                (,m2s ,m2))
            (make-writer (writer-value ,m2s)
                         (writer-compose (writer-fun ,m1s)
                                         (writer-fun ,m2s))))))
```

```
   ms
   :from-end t))
```

Macros WRITE!, WRITE-LIST!, RUN!, UNIT, FUNCALL!, PROGN! and LET! are implemented as a MACROLET defined by macro WITH-WRITER-MONAD.

```
(defmacro with-writer-monad (&body body)
  `(macrolet
       ((unit (a) (list 'writer-unit a))
        (funcall! (k m) (list 'writer-funcall! k m))
        (progn! (&body ms) (append '(writer-progn!) ms))
        (let! (decls m) (list 'writer-let! decls m))
        (write! (&body ws) (append '(writer-write!) ws))
        (write-list! (&body wss) (append '(writer-write-list!) wss))
        (run! (m) (list 'writer-run! m)))
     ,@body))
```

Now we can take our old example

```
  (with-writer-monad
    (let! ((x1 e1)
           (x2 e2))
          (progn! m1 m2
                  (unit (list x1 x2)))))
```

and look at its macro expansion.

```
  (LET* ((#:G1297 E1)
         (X1 (CAR #:G1297))
         (#:G1298
          (LET* ((#:G1295 E2)
                 (X2 (CAR #:G1295))
                 (#:G1296
                  (LET ((#:G1301 M1)
                        (#:G1302
                         (LET ((#:G1299 M2) (#:G1300 (CONS (LIST X1 X2)
NIL)))
                           (CONS (CAR #:G1300)
                                 (LET ((#:G1303 (CDR #:G1299))
                                       (#:G1304 (CDR #:G1300)))
                                   (IF (NULL #:G1304) (PROGN #:G1303)
                                       (IF (NULL #:G1303) (PROGN #:G1304)
                                           (THE T
                                             (PROGN
                                               #'(LAMBDA (X)
                                                   (FUNCALL #:G1303
                                                            (FUNCALL #:G1304

X)))))))))))
                    (CONS (CAR #:G1302)
                          (LET ((#:G1305 (CDR #:G1301)) (#:G1306 (CDR
#:G1302)))
                            (IF (NULL #:G1306) (PROGN #:G1305)
                                (IF (NULL #:G1305) (PROGN #:G1306)
```

```
                                    (THE T
                                      (PROGN
                                        #'(LAMBDA (X)
                                            (FUNCALL #:G1305
                                                     (FUNCALL #:G1306
                                                              X)))))))))))
            (CONS (CAR #:G1296)
                  (LET ((#:G1307 (CDR #:G1295)) (#:G1308 (CDR #:G1296)))
                    (IF (NULL #:G1308) (PROGN #:G1307)
                        (IF (NULL #:G1307) (PROGN #:G1308)
                            (THE T
                              (PROGN
                                #'(LAMBDA (X)
                                    (FUNCALL #:G1307
                                             (FUNCALL #:G1308 X)))))))))))
    (CONS (CAR #:G1298)
          (LET ((#:G1309 (CDR #:G1297)) (#:G1310 (CDR #:G1298)))
            (IF (NULL #:G1310) (PROGN #:G1309)
                (IF (NULL #:G1309) (PROGN #:G1310)
                    (THE T
                      (PROGN
                        #'(LAMBDA (X)
                            (FUNCALL #:G1309 (FUNCALL #:G1310 X)))))))))))
```

Although the expanded code looks long, it's straightforward enough. It mainly consists of the IF conditions and creations of the short-living CONS-pairs at each step. The anonymous functions are created only in case of need. The compiled code should be rather cheap. Moreover, it can be efficient if the compiler can optimize the short-living CONS-pairs.

The next example illustrates the use of the WITH-WRITER-MONAD macro.

```
(defun writer-test ()
  (destructuring-bind (a log)
      (with-writer-monad
        (run!
         (progn!
          (write! 1)
          (write! 2 3 4)
          (write-list! '(5 6))
          (write-list! '(7) '(8) '(9))
          (unit 'ok))))
    (format t "Computed value = ~a~%" a)
    (format t "Written log = ~a~%" log)))
```

This is its output.

```
CL-USER> (writer-test)
Computed value = OK
Written log = (1 2 3 4 5 6 7 8 9)
NIL
```

Monad Transformers

It's possible to create a macro analog of the monad transformer in Common Lisp. Such a macro must be parameterized and it must define macro LIFT! that has the same meaning as the *lift* function in Haskell.

```
class MonadTrans where
    lift :: (Monad m) => m a -> t m a
```

In the next sections are defined macros WITH-READER-MONAD-TRANS, WITH-WRITER-MONAD-TRANS and WITH-STATE-MONAD-TRANS. They are examples of the monad transformer macros. Each of them accepts the first parameter which must be a name of some monad macro in parentheses.

For example, we can write:

```
(with-reader-monad-trans (with-writer-monad)

  ;; It works within the WITH-READER-MONAD-TRANS macro

  (let! ((x (read!)))

        ;; It calls WRITE! within the WITH-WRITER-MONAD macro

        (lift!
         (with-writer-monad
           (write! x)))))
```

For this case we can create a separate monad macro and define the WRITE! macro on more high level using LIFT!

```
(defmacro with-reader-writer-monad (&body body)
  `(with-reader-monad-trans (with-writer-monad)
     (macrolet
         ((write! (&body bs)
            `(lift!
              (with-writer-monad
                (write! ,@bs)))))
       ,@body)))
```

The monad transformer macros can be nested.

```
(with-reader-monad-trans
    (with-writer-monad-trans
        (with-maybe-monad))

  (progn!

   ;; It evaluates (f x) within
   ;; the WITH-WRITER-MONAD-TRANS macro

   (lift!
    (with-writer-monad-trans (with-maybe-monad)
      (f x)))

   ;; It evaluates (g x) within
```

```
   ;; the WITH-MAYBE-MONAD macro

   (lift!
    (lift!
     (with-maybe-monad
       (g x))))))
```

The LIFT! macro must know a name of the inner monad macro to call the corresponded inner return and bind functions. This is a crucial point. It is applied to macros UNIT, FUNCALL!, PROGN! and LET! as well.

In the next sections you will see how the monad transformer macros can be implemented. All examples follow a common pattern.

```
(defmacro with-some-monad-trans
    (inner-monad &body body)

  `(with-monad-trans
      (with-some-monad-trans ,inner-monad)

     (macrolet
         ;; Definitions of UNIT, FUNCALL!, PROGN!, LET!
         ;; and possibly some other macros

         ,@body)))
```

Note how the definition of WITH-SOME-MONAD-TRANS recursively refers to itself. It is important.

WITH-MONAD-TRANS s a utility macro that allows the monad transformer implementer to use two auxiliary macros WITH-INNER-MONAD-TRANS and WITH-OUTER-MONAD-TRANS in accordance with the following scheme.

```
(with-some-monad-trans (with-inner-monad)

  ;; Here the WITH-SOME-MONAD-TRANS macro is active

  (with-inner-monad-trans (unique-id)

    ;; Here the WITH-INNER-MONAD macro is active, i.e.
    ;; a macro specified in the parameter

    (with-outer-monad-trans (unique-id)

      ;; Here the WITH-SOME-MONAD-TRANS macro
      ;; is active again

      ...)))
```

Here the WITH-INNER-MONAD-TRANS macro must precede the WITH-OUTER-MONAD-TRANS macro. UNIQUE-ID is some unique identifier which must be different for each occurrence. Usually, it is a generated value with help of function GENSYM.

This scheme allows the implementer to switch between the outer and inner monad macros.

The WITH-MONAD-TRANS macro has the following definition.

```
(defmacro with-monad-trans (outer-monad &body body)
  (let ((inner-monad (cadr outer-monad)))
    `(macrolet
         ((with-inner-monad-trans (id &body bs)
            (append '(with-inner-monad-prototype)
                    (list ',outer-monad)
                    (list ',inner-monad)
                    (list id)
                    bs))
          (with-outer-monad-trans (id &body bs)
            (append id bs)))
       ,@body)))
```

Please note that an implementation of the WITH-OUTER-MONAD-TRANS macro is common and doesn't depend on additional parameters, which allows us to switch to the outer monad even if case of the deep nested calls of WITH-MONAD-TRANS. The WITH-OUTER-MONAD-TRANS macro is expanded to a call of the macro represented by parameter *id*. The last macro must be already created by WITH-INNER-MONAD-PROTOTYPE before the inner monad macro is activated - this is why an order of precedence is important.

```
(defmacro with-inner-monad-prototype
    (outer-monad inner-monad id &body body)
  `(macrolet ((,@id (&body bs) (append ',outer-monad bs)))
     (,@inner-monad
      ,@body)))
```

The key point is that the WITH-INNER-MONAD-PROTOTYPE macro, i.e. WITH-INNER-MONAD-TRANS, creates a new macro that is expanded already to the outer monad macro, which name was passed as a parameter of WITH-MONAD-TRANS if you remember. The name of this new generated macro is defined by the value of parameter *id*. But WITH-OUTER-MONAD-TRANS macro has a common implementation and it is always expanded namely to that new macro, which is expanded in its turn to the outer monad macro regardless of that how deeply the WITH-MONAD-TRANS macros are nested, for the value of the *id* parameter is supposed to be unique.

It's worthy to note that if the monad macros consist of MACROLETs then macros WITH-MONAD-TRANS, WITH-INNER-MONAD-TRANS and WITH-OUTER-MONAD-TRANS add nothing but MACROLETs to the expanded code. Such a code should be rather efficient. All monad macros described in this article consist of MACROLETs only. It should be a general rule.

Nevertheless, in practice the Lisp compilers cannot process complex expressions, where the nested monad transformer macros are directly applied, although the simplest expressions are still compilable. There is a simple workaround for this problem. The approach is described in section Reducing Monad Macros.

Inner Monad Macros

In absence of the type classes in the language we have to distinguish somehow the operations performed in the inner and outer monads if we speak about the monad transformers. Now I will

introduce prototypes for macros INNER-UNIT, INNER-FUNCALL!, INNER-LET! and INNER-PROGN! that will be counterparts to macros UNIT, FUNCALL!, LET! and PROGN!. Only the first macros call the corresponded operations in the inner monad with one important exception. Their parameters are always evaluated lexically within the outer monad. It allows us to safely call these macros within the outer monad macro.

So, the INNER-UNIT macro prototype is as follows.

```
(defmacro generic-inner-unit (a)
  (let ((id (gensym)))
    `(with-inner-monad-trans (,id)
       (unit
        (with-outer-monad-trans (,id)
          ,a)))))
```

Please note that expression *a* is evaluated within the outer monad. It will be a general rule.

The INNER-FUNCALL! macro prototype is similar.

```
(defmacro generic-inner-funcall! (k m)
  (let ((id (gensym)))
    `(with-inner-monad-trans (,id)
       (funcall!
        (with-outer-monad-trans (,id) ,k)
        (with-outer-monad-trans (,id) ,m)))))
```

The INNER-LET! macro prototype is analogous.

```
(defmacro generic-inner-let! (decls m)
  (reduce
   #'(lambda (decl m)
       (destructuring-bind (x e) decl
         (let ((id (gensym)))
           `(with-inner-monad-trans (,id)
              (let! ((,x (with-outer-monad-trans (,id) ,e)))
                    (with-outer-monad-trans (,id) ,m))))))
   decls
   :from-end t
   :initial-value m))
```

Please note how carefully we restore the outer monad lexical context. It's very important. As we already discussed, it has no performance penalty for the generated code, although it creates a high load for the compiler because of numerous MACROLETs that are generated during the macro expansion.

The INNER-PROGN! macro prototype is more optimal.

```
(defmacro generic-inner-progn! (&body ms)
  (let ((id (gensym)))
    (let ((outer-ms (loop for m in ms collect
                          `(with-outer-monad-trans (,id) ,m))))
      `(with-inner-monad-trans (,id)
         (progn! ,@outer-ms)))))
```

Macros INNER-UNIT, INNER-FUNCALL!, INNER-LET! and INNER-PROGN! are implemented as a part of the MACROLET construct defined by macro WITH-MONAD-TRANS.

```
(defmacro with-monad-trans (outer-monad &body body)
  (let ((inner-monad (cadr outer-monad)))
    `(macrolet
         ((with-inner-monad-trans (id &body bs)
            (append '(with-inner-monad-prototype)
                    (list ',outer-monad)
                    (list ',inner-monad)
                    (list id)
                    bs))
          (with-outer-monad-trans (id &body bs)
            (append id bs))
          ;;
          (inner-unit (a) (list 'generic-inner-unit a))
          (inner-funcall! (k m) (list 'generic-inner-funcall! k m))
          (inner-progn! (&body ms) (append '(generic-inner-progn!) ms))
          (inner-let! (decls m) (list 'generic-inner-let! decls m)))
       ,@body)))
```

In most cases these new macros INNER-UNIT, UNNER-FUNCALL!, INNER-LET! and INNER-PROGN! cover all the needs and make low level macros WITH-INNER-MONAD-TRANS and WITH-OUTER-MONAD-TRANS unnecessary for the practical use in your code.

The Reader Monad Transformer

The Reader monad transformer is a parameterized version of the Reader monad but which can also act as the monad specified in the parameter. This is a very powerful abstraction. For example, we can combine the Reader monad transformer with the Writer monad. Then we can write a log and read an external value passed to the computation at the same time.

In Haskell the Reader monad transformer can be defined in the following way.

```
import Control.Monad
import Control.Monad.Trans

newtype ReaderTrans r m a =
    ReaderTrans {runReader :: r -> m a}

instance (Monad m) => Monad (ReaderTrans r m) where

    return a =
        ReaderTrans (\r -> return a)

    m >>= k =
        ReaderTrans (\r ->
                        do a <- runReader m r
                           let m' = k a
                           runReader m' r)

instance MonadTrans (ReaderTrans r) where
    lift m = ReaderTrans (\r -> m)
```

```
read :: (Monad m) => ReaderTrans r m r
read = ReaderTrans (\r -> return r)
```

Please note that the return and bind functions are mixed. Some of them are related to the ReaderTrans monad itself. Others are related already to the parameter monad *m*. It says that we need helper macros INNER-UNIT, INNER-FUNCALL!, INNER-LET! and INNER-PROGN! introduced above.

I'll define macro WITH-READER-MONAD-TRANS based on the WITH-MONAD-TRANS macro. Therefore the specified helper macros will be accessible.

The UNIT macro prototype uses INNER-UNIT.

```
(defmacro reader-trans-unit (a)
  (let ((r (gensym)))
    `#'(lambda (,r)
         (declare (ignore ,r))
         (inner-unit ,a))))
```

Please note that expression *a* is evaluated in the context of the WITH-READER-MONAD-TRANS macro, not in the context of the inner monad. It will be true for all next definitions as well.

The FUNCALL! macro prototype is also similar to its non-parameterized version.

```
(defmacro reader-trans-funcall! (k m)
  (let ((r (gensym))
        (a (gensym))
        (kg (gensym)))
    `#'(lambda (,r)
         (let ((,kg ,k))
           (inner-let! ((,a (funcall ,m ,r)))
                       (funcall (funcall ,kg ,a) ,r))))))
```

It corresponds to the definition written in Haskell. Only the order of parameters is different. Also all notes that I did for the non-parameterized version remain true. The generated code can be optimized by the compiler under some circumstances.

As before, the LET! macro prototype is more efficient.

```
(defmacro reader-trans-let! (decls m)
  (reduce #'(lambda (decl m)
              (destructuring-bind (x e) decl
                (let ((r (gensym)))
                  `#'(lambda (,r)
                       (inner-let! ((,x (funcall ,e ,r)))
                         (funcall ,m ,r))))))
          decls
          :from-end t
          :initial-value m))
```

We only replaced LET with INNER-LET! to take a value from the inner computation.

The PROGN! macro prototype uses INNER-PROGN! to bind the inner computations.

```
(defmacro reader-trans-progn! (&body ms)
  (reduce #'(lambda (m1 m2)
              (let ((r (gensym)))
                `#'(lambda (,r)
                     (inner-progn!
                      (funcall ,m1 ,r)
                      (funcall ,m2 ,r)))))
          ms
          :from-end t))
```

Being applied in complex nested expressions, all macros are expanded to a code that can be efficiently optimized by the compiler because of LAMBDAs and FUNCALLs that alternate with each other.

The READ! macro prototype uses already the INNER-UNIT macro to wrap the environment value in the inner monad.

```
(defmacro reader-trans-read! ()
  (let ((r (gensym)))
    `#'(lambda (,r)
         (inner-unit ,r))))
```

The RUN! macro prototype is the same, but now it returns a computation result wrapped in the inner monad.

```
(defmacro reader-trans-run! (m r)
  `(funcall ,m ,r))
```

So far, the macros defined replicate the interface of the WITH-READER-MONAD macro. Now I'll define the LIFT! macro that will allow us to perform any computation in the inner monad. This is namely that thing that allows the parameterized monad transformer to act as a monad specified in its parameter.

```
(defmacro reader-trans-lift! (m)
  (let ((r (gensym)))
    `#'(lambda (,r)
         (declare (ignore ,r))
         ,m)))
```

Macros LIFT!, READ!, UNIT, FUNCALL!, LET! and PROGN! are implemented as a MACROLET defined by the WITH-READER-MONAD-TRANS macro, which in its turn follows a common pattern described in section Monad Transformers.

```
(defmacro with-reader-monad-trans (inner-monad &body body)
  `(with-monad-trans (with-reader-monad-trans ,inner-monad)
     (macrolet
         ((unit (a) (list 'reader-trans-unit a))
          (funcall! (k m) (list 'reader-trans-funcall! k m))
          (progn! (&body ms) (append '(reader-trans-progn!) ms))
          (let! (decls m) (list 'reader-trans-let! decls m))
          (read! () (list 'reader-trans-read!))
          (run! (m r) (list 'reader-trans-run! m r))
          (lift! (m) (list 'reader-trans-lift! m)))
       ,@body)))
```

Now the monad macro generates much code. Even after removing the MACROLETs that mean nothing for the execution time but that may slow down the compilation process, the macro expansion may be still long depending on the specified inner monad. Therefore I will use a simpler example to illustrate the code generation.

```
(with-reader-monad-trans (with-maybe-monad)
  (let! ((x e)) m))
```

After removing all the MACROLETs (with help of CLISP), the code expansion looks like

```
#'(LAMBDA (#:G4207)
          (LET ((#:G4209 (FUNCALL E #:G4207))) (IF (NULL #:G4209) NIL (LET
((X (CAR #:G4209))) (FUNCALL M #:G4207)))))
```

Here is a test of the monad macro.

```
(defun reader-trans-test ()
  (destructuring-bind (a log)

      (with-writer-monad
        (run!

         (with-reader-monad-trans (with-writer-monad)
           (run!

            (let! ((x (read!)))
                  (progn!

                   (lift!
                    (with-writer-monad
                      (write! x)))

                   (unit 'ok)))
            10))))

    (format t "Computed value=~a~%" a)
    (format t "Written log=~a~%" log)))
```

This is its output.

```
CL-USER> (reader-trans-test)
Computed value=OK
Written log=(10)
NIL
```

The State Monad Transformer

The State monad transformer is a parameterized version of the State monad but which can also behave like a monad specified in the parameter. For example, we can create a version of the State monad transformer parameterized by the Writer monad. Then we can manage the state and write a log during the computation simultaneously.

I'll use the following definition of the State monad transformer written in Haskell.

```
import Control.Monad
import Control.Monad.Trans

newtype StateTrans st m a = StateTrans {runState :: st -> m (a, st)}

instance (Monad m) => Monad (StateTrans st m) where

    return a = StateTrans (\st -> return (a, st))

    m >>= k = StateTrans (\st ->
                              do (a, st') <- runState m st
                                 let m' = k a
                                 runState m' st')

instance MonadTrans (StateTrans st) where
    lift m = StateTrans (\st -> do a <- m; return (a, st))

get :: (Monad m) => StateTrans st m st
get = StateTrans (\st -> return (st, st))

put :: (Monad m) => st -> StateTrans st m ()
put st' = StateTrans (\_ -> return ((), st'))
```

We see that the return and bind functions are mixed as it was in case of the Reader monad transformer. Some functions correspond to the StateTrans monad. Others correspond to the inner monad *m*. Hence we need macros INNER-UNIT, INNER-FUNCALL!, INNER-LET! and INNER-PROGN! provided by the WITH-MONAD-TRANS macro.

I'll define a new macro WITH-STATE-MONAD-TRANS based on WITH-MONAD-TRANS in accordance with the general pattern described in section Monad Transformers. Also the new macro will be similar to its non-parameterized counterpart WITH-STATE-MONAD. The WITH-STATE-MONAD-TRANS macro will define macros GET!, PUT! and RUN!. Only the latter will return a value wrapped in the inner monad.

The UNIT macro prototype is similar but it uses INNER-UNIT to wrap a pair in the inner monad.

```
(defmacro state-trans-unit (a)
  (let ((st (gensym)))
    `#'(lambda (,st)
         (inner-unit
          (make-state ,a ,st)))))
```

As before, expression *a* is evaluated inside LAMBDA, i.e. the evaluation is delayed. This strategy will be applied to all other macros defined further.

The FUNCALL! macro prototype is similar too, but now it uses INNER-LET! to get a raw value from the inner monad.

```
(defmacro state-trans-funcall! (k m)
  (let ((st (gensym))
        (p (gensym))
        (a (gensym))
```

```
        (kg (gensym)))
    `#'(lambda (,st)
         (let ((,kg ,k))
           (inner-let! ((,p (funcall ,m ,st)))
             (let ((,a (state-value ,p)))
               (funcall (funcall ,kg ,a)
                        (state-state ,p))))))))
```

The notes that I did earlier for the State monad are applicable now as well. Expression *m* is used as the first argument of the FUNCALL function. This expression is a monad value, i.e. an anonymous function. If the s-expression for *m* is available then *m* will be expanded to the LAMBDA expression. These LAMBDA and FUNCALL can be reduced by the smart compiler.

As usual, the LET! macro prototype generates a more efficient code than FUNCALL!.

```
(defmacro state-trans-let! (decls m)
  (reduce #'(lambda (decl m)
              (destructuring-bind (x e) decl
                (let ((st (gensym))
                      (p (gensym)))
                  `#'(lambda (,st)
                       (inner-let! ((,p (funcall ,e ,st)))
                         (let ((,x (state-value ,p)))
                           (funcall ,m (state-state ,p))))))))
          decls
          :from-end t
          :initial-value m))
```

Here expressions *e* and *m* are monad values, i.e. anonymous functions. Moreover, if we create a multi-level LET! expression then the s-expression for *m* is available for all cases but probably the last. This s-expression is started with LAMBDA. Therefore LAMBDAs and FUNCALLs can be reduced by the compiler too.

The PROGN! macro prototype doesn't bind variables but it passes the state through the computation like the previous macros.

```
(defmacro state-trans-progn! (&body ms)
  (reduce #'(lambda (m1 m2)
              (let ((st (gensym))
                    (p (gensym)))
                `#'(lambda (,st)
                     (inner-let! ((,p (funcall ,m1 ,st)))
                       (funcall ,m2 (state-state ,p))))))
          ms
          :from-end t))
```

The RUN! macro launches a computation specified in the first parameter. The second parameter defines an initial state. The macro returns a list wrapped in the inner monad. The first value of the list is a result of the computation itself. The second value is a final state.

```
(defmacro state-trans-run! (m init-st)
  (let ((p (gensym)))
```

```
    `(inner-let! ((,p (funcall ,m ,init-st)))
       (inner-unit
        (list (state-value ,p)
              (state-state ,p))))))
```

The GET! macro prototype returns the current state wrapped in the outer monad.

```
(defmacro state-trans-get! ()
  (let ((st (gensym)))
    `#'(lambda (,st)
         (inner-unit
          (make-state ,st ,st)))))
```

The PUT! macro prototype has one parameter that specifies a new value for the state. It allows us to modify the state. The new value will be then passed to the rest part of the computation. The macro returns NIL wrapped in the outer monad.

```
(defmacro state-trans-put! (new-st)
  (let ((st (gensym)))
    `#'(lambda (,st)
         (declare (ignore ,st))
         (inner-unit
          (make-state nil ,new-st)))))
```

The LIFT! macro endows the parameterized monad transformer with an ability to act as a monad specified in the parameter. The macro accepts any computation in the inner monad. This inner computation becomes a part of the outer computation.

```
(defmacro state-trans-lift! (m)
  (let ((st (gensym))
        (a (gensym)))
    `#'(lambda (,st)
         (inner-let! ((,a ,m))
           (inner-unit
            (make-state ,a ,st))))))
```

Macros GET!, PUT!, RUN!, LIFT!, UNIT, FUNCALL!, LET! and PROGN! are implemented as a MACROLET defined by the WITH-STATE-MONAD-TRANS macro that follows a common pattern of the monad transformer macros.

```
(defmacro with-state-monad-trans (inner-monad &body body)
  `(with-monad-trans (with-state-monad-trans ,inner-monad)
     (macrolet
         ((unit (a) (list 'state-trans-unit a))
          (funcall! (k m) (list 'state-trans-funcall! k m))
          (progn! (&body ms) (append '(state-trans-progn!) ms))
          (let! (decls m) (list 'state-trans-let! decls m))
          (get! () (list 'state-trans-get!))
          (put! (new-st) (list 'state-trans-put! new-st))
          (run! (m init-st) (list 'state-trans-run! m init-st))
          (lift! (m) (list 'state-trans-lift! m)))
       ,@body)))
```

The code generation can be illustrated on the following example.

```
(with-state-monad-trans (with-maybe-monad)
  (let! ((x e)) m))
```

After removing all MACROLETs (with help of CLISP), the code is expanded to

```
#'(LAMBDA (#:G4372)
    (LET ((#:G4375 (FUNCALL E #:G4372)))
      (IF (NULL #:G4375) NIL (LET ((#:G4373 (CAR #:G4375))) (LET ((X (CAR
#:G4373))) (FUNCALL M (CDR #:G4373)))))))
```

The next test enumerates all items of the tree. It creates a new tree, where each item is replaced with a CONS-pair, consisting of the item itself and its sequence number. Also the test function saves all enumerated items in the list and shows it as a log.

```
(defun state-trans-test (tree)
  (labels
      ((order (tree)
         (with-state-monad-trans (with-writer-monad)
           (cond ((null tree) (unit nil))
                 ((consp tree)
                  (let! ((t1 (order (car tree)))
                         (t2 (order (cdr tree))))
                    (unit (cons t1 t2))))
                 (t
                  (let! ((n (get!)))
                    (let ((new-n (+ n 1)))
                      (progn!

                       (lift!
                        (with-writer-monad
                          (write! tree)))

                       (put! new-n)
                       (unit (cons tree new-n)))))))))))

    (destructuring-bind ((new-tree new-state) saved-log)
        (with-writer-monad
          (run!
           (with-state-monad-trans (with-writer-monad)
             (run! (order tree) 0))))

      (format t "Item count=~a~%" new-state)
      (format t "New tree=~a~%" new-tree)
      (format t "Written log=~a~%" saved-log))))
```

Now we can launch a test.

```
CL-USER> (state-trans-test '(5 4 (1 2 (3))))
Item count=5
New tree=((5 . 1) (4 . 2) ((1 . 3) (2 . 4) ((3 . 5))))
```

```
Written log=(5 4 1 2 3)
NIL
```

The Writer Monad Transformer

The Writer monad transformer is a parameterized version of the Writer monad but which can also act as a monad specified in the parameter. For example, we can parameterize this transformer by the Maybe monad. As a result, we'll receive a new monad that will allow us to write a log and cut all computations immediately in case of need.

I will use the next definition written in Haskell.

```
import Control.Monad
import Control.Monad.Trans

newtype WriterTrans w m a = WriterTrans (m (a, [w] -> [w]))

runWriter :: (Monad m) => WriterTrans w m a -> m (a, [w])
runWriter (WriterTrans m) = do (a, f) <- m
                               return (a, f [])

write :: (Monad m) => w -> WriterTrans w m ()
write w = WriterTrans (return ((), \xs -> w : xs))

writeList :: (Monad m) => [w] -> WriterTrans w m ()
writeList ws = WriterTrans (return ((), \xs -> ws ++ xs))

instance (Monad m) => Monad (WriterTrans w m) where

    return a = WriterTrans (return (a, id))

    (WriterTrans m) >>= k =
        WriterTrans (do (a, f) <- m
                        let WriterTrans m' = k a
                        (a', f') <- m'
                        return (a', f . f'))

instance MonadTrans (WriterTrans w) where
    lift m = WriterTrans (do a <- m; return (a, id))
```

As in case of the Reader monad transformer we can see a lot of the mixed functions return and bind. Some of them are related to *WriterTrans*. Others are related to monad *m*. Therefore we need again the WITH-MONAD-TRANS macro that contains definitions of INNER-UNIT, INNER-LET!, INNER-FUNCALL! and INNER-PROGN! that allow us to work with the parameter monad.

So, I'll define macro WITH-WRITER-MONAD-TRANS that will be based on the WITH-MONAD-TRANS macro in accordance with the general pattern described in section Monad Transformers. This new macro will be similar to the WITH-WRITER-MONAD macro. It will be only parameterized and it will also contain macro LIFT!, an analog of the *lift* function from Haskell.

The WRITE! macro uses now the INNER-UNIT macro as we have to wrap a CONS-pair created with help of MAKE-WRITER.

```
(defmacro writer-trans-write! (&body ws)
    (if (= 1 (length ws))
        ;; An optimized case
        (let ((w (nth 0 ws))
              (v (gensym)))
          `(inner-unit
            (make-writer nil
                         (let ((,v ,w))
                           #'(lambda (xs) (cons ,v xs))))))
        ;; A general case
        (let ((vs (gensym)))
          `(inner-unit
            (make-writer nil
                         (let ((,vs (list ,@ws)))
                           #'(lambda (xs)
                               (append ,vs xs))))))))
```

The WRITE-LIST! macro prototype is similar. It also returns NIL in the outer monad.

```
(defmacro writer-trans-write-list! (&body wss)
    (if (= 1 (length wss))
        ;; An optimized case
        (let ((ws (nth 0 wss))
              (vs (gensym)))
          `(inner-unit
            (make-writer nil
                         (let ((,vs ,ws))
                           #'(lambda (xs) (append ,vs xs))))))
        ;; A general case
        (let ((vss (gensym)))
          `(inner-unit
            (make-writer nil
                         (let ((,vss (list ,@wss)))
                           #'(lambda (xs)
                               (reduce #'append ,vss
                                       :from-end t
                                       :initial-value xs))))))))
```

Please note that in the both macros we evaluate the values *ws* and *wss* first and then return new functions. The real writing operation will be delayed.

Now the RUN! macro returns a list of two values, where the list is wrapped in the inner monad. The first value of the list is a result of the computation. The second value is a log written during this computation.

```
(defmacro writer-trans-run! (m)
  (let ((x (gensym))
        (fun (gensym)))
    `(inner-let! ((,x ,m))
                 (inner-unit
                  (list (writer-value ,x)
                        (let ((,fun (writer-fun ,x)))
                          (if (not (null ,fun))
                              (funcall ,fun nil))))))))
```

The UNIT macro prototype also uses the INNER-UNIT macro to wrap a value in the inner monad.

```
(defmacro writer-trans-unit (a)
  `(inner-unit
    (make-writer ,a nil)))
```

Please note that expression *a* is expanded within the outer monad macro, i.e. within WITH-WRITER-MONAD-TRANS, for which the INNER-UNIT macro is responsible.

The FUNCALL! macro prototype is also similar to its non-parameterized counterpart.

```
(defmacro writer-trans-funcall! (k m)
  (let ((ks (gensym))
        (ms (gensym))
        (a (gensym))
        (ka (gensym)))
    `(let ((,ks ,k))
       (inner-let! ((,ms ,m))
         (let ((,a (writer-value ,ms)))
           (inner-let! ((,ka (funcall ,ks ,a)))
             (inner-unit
              (make-writer (writer-value ,ka)
                           (writer-compose (writer-fun ,ms)
                                           (writer-fun ,ka)))))))))))
```

As usual, the LET! macro prototype is more optimal.

```
(defmacro writer-trans-let! (decls m)
  (reduce
   #'(lambda (decl m)
       (destructuring-bind (x e) decl
         (let ((es (gensym))
               (ms (gensym)))
           `(inner-let! ((,es ,e))
              (let ((,x (writer-value ,es)))
                (inner-let! ((,ms ,m))
                  (inner-unit
                   (make-writer (writer-value ,ms)
                                (writer-compose (writer-fun ,es)
                                                (writer-fun ,ms)))))))))))
   decls
   :from-end t
   :initial-value m))
```

The PROGN! macro prototype was also slightly modified.

```
(defmacro writer-trans-progn! (&body ms)
  (reduce
   #'(lambda (m1 m2)
       (let ((m1s (gensym))
             (m2s (gensym)))
         `(inner-let! ((,m1s ,m1)
                       (,m2s ,m2))
            (inner-unit
```

```
            (make-writer (writer-value ,m2s)
                         (writer-compose (writer-fun ,m1s)
                                         (writer-fun ,m2s)))))))
   ms
   :from-end t))
```

As in case of the Reader monad transformer macro we can define the LIFT! macro that will allow us to perform any computation in the inner monad. This is that thing that allows the parameterized monad transformer to act as a monad specified in its parameter.

```
(defmacro writer-trans-lift! (m)
  (let ((a (gensym)))
    `(inner-let! ((,a ,m))
       (inner-unit
        (make-writer ,a nil)))))
```

Macros LIFT!, WRITE!, WRITE-LIST!, UNIT, FUNCALL!, LET! and PROGN! are implemented as a MACROLET defined by macro WITH-WRITER-MONAD-TRANS, which in its turn follows a common pattern of the monad transformer macros.

```
(defmacro with-writer-monad-trans (inner-monad &body body)
  `(with-monad-trans (with-writer-monad-trans ,inner-monad)
     (macrolet
         ((unit (a) (list 'writer-trans-unit a))
          (funcall! (k m) (list 'writer-trans-funcall! k m))
          (progn! (&body ms) (append '(writer-trans-progn!) ms))
          (let! (decls m) (list 'writer-trans-let! decls m))
          (write! (&body ws) (append '(writer-trans-write!) ws))
          (write-list! (&body wss) (append '(writer-trans-write-list!) wss))
          (run! (m) (list 'writer-trans-run! m))
          (lift! (m) (list 'writer-trans-lift! m)))
       ,@body)))
```

This monad macro generates a lot of MACROLETs. They don't impact the performance of the executed code, although the compilation becomes a more difficult task for the Lisp system.

Let's take the following sample

```
(with-writer-monad-trans
    (with-reader-monad-trans
        (with-maybe-monad))

  (let! ((x e)) m))
```

After removing the MACROLETs (with help of CLISP) the macro expansion looks like this

```
#'(LAMBDA (#:G4325)
    (LET ((#:G4327 (FUNCALL E #:G4325)))
      (IF (NULL #:G4327) NIL
          (LET ((#:G4322 (CAR #:G4327)))
            (FUNCALL
             (LET ((X (CAR #:G4322)))
```

```
                #'(LAMBDA (#:G4329)
                    (LET ((#:G4331 (FUNCALL M #:G4329)))
                      (IF (NULL #:G4331) NIL
                          (LET ((#:G4323 (CAR #:G4331)))
                            (FUNCALL
                             #'(LAMBDA (#:G4333)
                                 (DECLARE (IGNORE #:G4333))
                                 (CONS
                                  (CONS (CAR #:G4323)
                                        (LET ((#:G4335 (CDR #:G4322)) (#:G4336
(CDR #:G4323)))
                                          (COND ((NULL #:G4336) #:G4335)
((NULL #:G4335) #:G4336)
                                                (T
                                                 #'(LAMBDA (X)
                                                     (FUNCALL #:G4335 (FUNCALL
#:G4336 X)))))))
                                  NIL))
                             #:G4329))))))
             #:G4325)))))
```

Here is a test.

```
(defun writer-trans-test ()
  (let ((m (with-writer-monad-trans (with-maybe-monad)

             (run!
              (progn!
               (write! 1)
               (write! 2 3)
               (lift! (make-maybe))    ; FAIL
               (write-list! '(4 5 6))
               (unit 'ok))))))

    (if (maybe-just-p m)

        (progn
          (destructuring-bind (a log) (maybe-just m)
            (format t "Computed value=~a~%" a)
            (format t "Written log=~a~%" log)))

        (format t "Computation was interrupted~%"))))
```

If you'll try to compile it with help of SBCL, then the compiler will warn about an unreachable code!

This is an output of the test.

```
CL-USER> (writer-trans-test)
Computation was interrupted
NIL
```

Reducing Monad Macros

The ordinary monad macros are expanded to a construct that contains a single MACROLET. Therefore the expressions that use these monad macros are compiled fast. The monad macros built on the monad transformers are not that case. They are expanded already to a construct that may contain a lot of nested MACROLETs. It becomes a real problem for the Lisp compiler. Not any expression consisting of the nested monad transformer macros can be even compiled!

Below is described an approach that allows the Lisp system to compile monad transformer macros of any complexity and to do it relatively fast. The main idea is to replace the macros with functions. The drawback of this method is that an executable code becomes a little bit slower than it could be in case of the pure macro expansion.

I'll illustrate the method on the parameterized twice macro WITH-WRITER-MONAD-TRANS (WITH-READER-MONAD-TRANS (WITH-MAYBE-MONAD)).

First, we create a short name for our source macro, lifting the READ! macro from the Writer monad transformer.

```
(defmacro with-opt-proto (&body body)
  `(with-writer-monad-trans
       (with-reader-monad-trans
           (with-maybe-monad))
     (macrolet
         ((read! ()
            `(lift!
              (with-reader-monad-trans
                  (with-maybe-monad)
                (read!)))))
       ,@body)))
```

This new macro provides macros READ!, WRITE!, WRITER-LIST!, RUN!, LIFT!, UNIT, FUNCALL!, LET! and PROGN!. Now we'll create functions for them, i.e. all macros will be expanded only once.

```
(defun opt-read! ()
  (with-opt-proto
    (read!)))

(defun opt-write! (&rest ws)
  (with-opt-proto
    (if (= 1 (length ws))
        (write! (nth 0 ws))
        (write-list! ws))))

(defun opt-write-list! (&rest wss)
  (with-opt-proto
    (if (= 1 (length wss))
        (write-list! (nth 0 wss))
        (reduce #'(lambda (ws m)
                    (progn! (write-list! ws) m))
                wss
                :from-end t
                :initial-value (unit nil)))))
```

In the last function we create a sequence of the computations and always return NIL wrapped in the monad.

The top level RUN! macro returns a list wrapped in the inner monad WITH-READER-MONAD-TRANS (WITH-MAYBE-MONAD). This list contains two values. The first is a result of the computation. The second value is a log written during this computation. The inner RUN! macro returns already a value in the Maybe monad. Therefore we can unite two RUN! macros and return the list of two values in the Maybe monad.

```
(defun opt-run! (m r)
  (with-reader-monad-trans (with-maybe-monad)
    (run! (with-opt-proto
            (run! m))
          r)))
```

We also have two LIFT! macros. We can unite them too. We pass some computation in the Maybe monad, for example, a value created with help of the MAKE-MAYBE function, and the new function returns the corresponded computation wrapped in the outer monad WITH-OPT-PROTO.

```
(defun opt-lift! (m)
  (with-opt-proto
    (lift!
      (with-reader-monad-trans (with-maybe-monad)
        (lift! m)))))
```

Now we can define the return and bind functions.

```
(defun opt-unit (a)
  (with-opt-proto
    (unit a)))

(defun opt-funcall! (k m)
  (with-opt-proto
      (funcall! k m)))
```

We have all functions to define a new monad macro with help of the WITH-MONAD macro. I'll call this new monad macro a *reduction form* of the source macro. It contains only two nested MACROLETs, which makes the code with the new macro easily compilable regardless of that how complex are the expressions built with help of macros UNIT, FUNCALL!, LET! and PROGN!.

```
(defmacro with-opt-monad (&body body)
  `(with-monad (opt-unit opt-funcall!)
     (macrolet
         ((read! () `(opt-read!))
          (write! (&body bs) `(opt-write! ,@ bs))
          (write-list! (&body bs) `(opt-write-list! ,@bs))
          (run! (m r) `(opt-run! ,m ,r))
          (lift! (m) `(opt-lift! ,m)))
       ,@body)))
```

In difficult cases the reduction can be applied many times. For example, to receive a monad macro with the same behavior, we could first reduce macro WITH-READER-MONAD-TRANS (WITH-MAYBE-MONAD) to new macro WITH-READER-MAYBE-MONAD. Then we could reduce macro WITH-WRITER-MONAD-TRANS (WITH-READER-MAYBE-MONAD) to form WITH-ALTOPT-MONAD, which would be equivalent to the WITH-OPT-MONAD macro. Only the more reduction steps we apply the less efficient code is generated by the compiler. But sometimes the reduction is a single possible way to make the code compilable.

This is a small test with the new monad.

```
(defun opt-test ()
  (let ((m (with-opt-monad
             (run!
              (progn!

               (write! 1)
               (write! 2 3)
               (write-list! '(4 5 6))

               (let! ((x (read!)))
                 (lift! (make-maybe :just x))))

              10))))

    (if (maybe-just-p m)

        (progn
          (destructuring-bind (a log) (maybe-just m)
            (format t "Computed value=~a~%" a)
            (format t "Written log=~a~%" log)))

        (format t "Computation was interrupted~%"))))
```

The test returns the following results.

```
CL-USER> (opt-test)
Computed value=10
Written log=(1 2 3 4 5 6)
NIL
```

Loops

The monad macros can perfectly coexist with the standard constructions IF, COND, PROGN, LET, LET*, FLET, LABELS, MACROLET, SYMBOL-MACROLET, LAMBDA, FUNCALL, DESTRUCTURING -BIND and some others in one expression. On the contrary, the standard loop macros DO, DOLIST, DOTIMES and LOOP are not so simple. If we perform monad computations in some loop then, generally speaking, we have to connect all the intermediate monad computations into one with help of something like the PROGN! macro. This is a key point.

I won't dwell on this subject, but I want to say that some monad macros could be implemented as a MACROLET defining macros DO!, DOLIST! and DOTIMES! that would be monadic counterparts to the standard loop macros. Here we would probably have to add some monad representation of an empty

loop, i.e. an empty monad computation. It could be a macro named ZERO!, for example. Also I think that the LOOP macro is more difficult case and I'm not sure that a monadic counterpart can be created for it.

Other Monad Macros

In Haskell we can define a small set of polymorphic functions that will work with any monad. Here in Common Lisp we can partially implement the same idea but in another way. Taking into account that the number of such common functions is relatively small and they are usually simple, we can try to implement them with help of a MACROLET that would be supplied together with the monad macro like WITH-MONAD.

In general case we can define a prototype for the functor map function, which I'll call FMAP.

```
(defmacro generic-fmap (f m)
   ;; an analog of the fmap function from Haskell
   (let ((fun (gensym))
         (x (gensym)))
      `(let ((,fun ,f))
            (let! ((,x ,m))
                  (unit (funcall ,fun ,x))))))
```

It's obvious that in case of the List monad the following definition will be much more efficient:

```
(defmacro list-fmap (f m)
   ;; fmap for the List monad
   `(mapcar ,f ,m))
```

It's easy to provide each monad macro with its own optimized version of the FMAP macro. Moreover, such a technique has no almost performance overhead.

The approach can be generalized for other monad functions. But the task of their creation deserves a separate article. Now I will only provide a naïve non-optimized version of another useful macro LIST!, which is an expanded version of the *sequence* function from Haskell

```
(defmacro list! (&body ms)
   (reduce
      #'(lambda (x xs)
           (let ((y (gensym))
                 (ys (gensym)))
              `(let! ((,y ,x)
                      (,ys ,xs))
                     (unit (cons ,y ,ys)))))
      ms
      :from-end t
      :initial-value (unit ())))
```

In case of the WITH-IDENTITY-MONAD macro the LIST! macro can be replaced with function LIST, which corresponds to the rule of thumb.

Conclusion

I tried to introduce the monads in the Lisp Way. I know that there were other attempts. They are mainly based on using generic functions that allow the programmer to write a polymorphic code but at the cost of some lost of the performance. My approach, on the contrary, allows the Lisp compiler to generate an efficient code but it lacks some flexibility.

Also I think that my approach is somewhere similar to the F# *workflows*. Only the monad macros play a role of the workflow builders.